Mark Palladino, Joshua Navarro (and originally Darrion Burgess, but he dropped)
CMPS 109 Fall 2016
MIS Project Phase 2 Report

Build instructions:

run “make client” to build client_misvm
run “make server” to build server_misvm
run “make clean” to remove misvm files and all .out and .err files

NOTE: we both worked on Linux, not Mac

Design Decisions (changes to part1):

What follows is all the **changes to old** parts of the program and our design decisions for how we implemented each part.

TempLoadVars struct instead of passing separately

Instead of passing all the temporary variables necessary for loading in function arguments, we made a struct called TempLoadVars and passed a pointer to that struct as 1 argument. We did this to reduce code verbosity.

Link instead of LinkJumps

We made the function linkJumps() into link() because we now had to link THREAD_BEGINs to THREAD_ENDs, make sure there’s at least 1 LOCK for every UNLOCK and vice versa, and make sure JMPs don’t jump outside of the thread they’re jumping from.

Variables as a map instead of a vector

We made the variables vector<Parameter*> in MISProgram into a map<string,Parameter*> where the string is the name of the variable and the Parameter* is the pointer to it. This way constants don’t waste space by having names. In addition, it speeds up by O(log(n)) the looking up variables to see if they exist (like when linking parameters of instructions and LOCKs and UNLOCKs to existing variables).

Design Decisions (additions):

What follows is all the **newly added** parts of the program and our design decisions for how we implemented each part.

Instructions have a pointer to the MISProgram

We made instructions have a pointer to the MISProgram that is running the instruction because we needed a way for the OUT instruction to output to the outBuffer in MISProgram. We also needed a way for the thread spawned by the THREAD_BEGIN instruction to be able to reference the instructions vector in MISProgram to execute them. We also needed a way for LOCKs and UNLOCKs to reference the variableLocks in MISProgram. Lastly, we needed a way for BARRIERs to reference the ThreadManager object in MISProgram (ThreadManager is discussed below). We solved these problems by making all instructions have a pointer to the MISProgram that is running it. While this wastes space on the instructions that don't use it, it's necessary if we are to maintain encapsulation.

VariableLocks

We made a struct for variable locks that are a mutex and an int representing the lock and the thread that acquired it. This is so that we can set the int whenever a LOCK instruction is done and make sure in an UNLOCK that the thread that LOCK'd it is the one that UNLOCK'd it. We use a struct instead of a class because the amount of code and complexity that would save compared to how much it would add made it seem not worth it.

Out and Err Output

For out and err output we added a string and a mutex for both the out and err buffers to be locked, filled, then unlocked while the server is handling clients. Then these buffers are sent back to the clients for them to write to the corresponding files. We used a string for the same reason we use it everywhere else in our program; it's very simple to use, reduces complexity, and reuses the already made string functions. We didn't make a separate class such as a OutputBuffer class because the amount of code and complexity that would save compared to how much it would add made it seem not worth it.

Connection, ChildThread and ThreadManager

We used the example code given in class for the Connection and GarbageCollector classes and made a few tweaks to let them better fit our project. The primary change we made was we removed the reference to the "nextConnection" from the Connection class and made the GarbageCollector class (renamed to ThreadManager) keep track of the next thread to manage. We did this because we also made a ChildThread class for threads that are spawned from the THREAD_BEGIN instruction and wanted to have the MISProgram have a ThreadManager that we can just add the ChildThread to and not have to worry about. And if we're going to make a ChildThread class, repeating the same code for the "nextChildThread" as in Connection seemed like bad design because it adds unnecessary complexity for extending the possible types of Threads that can be managed by the ThreadManager class.

Thread, TCPSocket and TCPServerSocket

We used the given Thread, TCPSocket, and TCPServerSocket helper code because it did everything we wanted, so why remake something that already is working well? However, we did make a few small changes. In TCPSocket, in readFromSocketWithTimeout(), we changed recv() to reuse the readFromSocket() code because we want to memset the buffer to 0 and make sure the buffer isn't NULL. In Thread, we added extra error handling and made the run() function explicitly call cleanup() after threadMainBody() instead of calling pthread_cleanup_push() because we used both used Linux.

Clientmain and Servermain

Because the client and server use a lot of the same code, we simply made two different main functions instead of two whole different project directories and everything. This made things a lot simpler by allowing us to keep all the code that was common the client and server in one place and keep all the code that is different, separate. We considered making a MISClient class and a MISServer class, but those classes would be very specific and not generic because we need them to do the very specific task of loading a file and sending it to a server then waiting for the server to process the code in a very specific way and send the results. So, we decided just to make all the core functionality of the possible MISClient and MISServer classes in the two main functions.

Verification and sending of the code

To verify code is correct before sending it to the server, we used the same load() function that we made in the first phase (with the added instructions). While this may be a bit inefficient because we're allocating variables and instructions we'll never use, it greatly reduces the complexity that necessarily comes with creating two separate functions validate() and load() that would be pretty much the same code in two places anyway.

Client Server Communication (TCP vs UDP and more)

We chose TCP over UDP because for this assignment we need to send a possibly large file over the Internet, make sure it arrives, and make sure it arrives in order, which is what TCP automatically does. If we chose UDP we would basically have to remake TCP. As for specifically how we communicate from client to server and vice versa, first the client sends the code in 65000 byte chunks and finishes the code with a semicolon. Then the client waits on the server to respond with the out and err (both ending with a semicolon). The server uses readFromSocketWithTimeout() because a malicious client could just make a bunch of connections without actually sending code and it would clog up the server with a bunch of Connection threads waiting on data that will never be sent. The client uses readFromSocket() because the client can always ctrl+C out if there's a server error, plus the client may send a lot of SLEEP instructions which will make having a timeout very complicated.

THREAD_BEGIN and THREAD_END

How: The way we implemented THREAD_BEGIN and THREAD_END is we added 2 NUMERIC constants to the THREAD_BEGIN instruction's parameters, one for the index of the THREAD_BEGIN instruction itself and one for the index of the THREAD_END instruction. These parameters are added in the Instruction's parseParameters() function. They are set in the MISProgram's link() function. Then when executing a THREAD_BEGIN instruction, we start a new ChildThread, passing the THREAD_BEGIN instruction to the object so that it can loop through the MISProgram's instructions vector starting at the instruction after THREAD_BEGIN until the instruction before THREAD_END. Also, we made THREAD_BEGIN basically act like a JMP by adding the same parameters as a vanilla JMP and setting the JMPFlag whenever it is hit. We set the instruction to JMP to as the THREAD_END's index + 1 so that the main thread jumps over the code dedicated to that thread.

Why: We implemented it this way because it seems like the best approach considering we need to achieve the functionality of 1) making the main thread jump over the thread's instructions and 2) making the thread loop through and execute only the instructions within the THREAD_BEGIN and THREAD_END instructions. This also seems to be the fastest approach as it doesn't require changing up the instructions vector and access of the instructions is still O(1), it also allows us to reuse the JMP code.

LOCK and UNLOCK

How: The way we implemented LOCK and UNLOCK is we first make sure that the referenced variable exists, and if it does, we either make or reference (depending on it's been made yet) a VariableLock. In addition to adding the index as a parameter, we also add the threadLevel of the LOCK and UNLOCK (threadLevel is just an int representing which thread we're in). Then when executing we just lock on the VariableLock's mutex and either set or check the threadLevel based on if it's a LOCK or UNLOCK. For LOCK we set it to the LOCK's threadLevel, for UNLOCK we make sure the VariableLock's last set threadLevel matches the UNLOCK's threadLevel.

Why: We add an index to the VariableLock because that way it references the VariableLock by O(1). We add the threadLevel because that way we can make sure the same thread the LOCKs it is the same that UNLOCKs it.

BARRIER

How: The way we implemented BARRIER is we just called the terminate() function of the ThreadManager in MISProgram that manages the ChildThreads spawned by THREAD_BEGIN.

Why: We did it this way because it is the most simple and efficient way because it reuses code that fits the goal of BARRIER perfectly.